设备树是为了将对板级的描述从内核中剔除而引入的，引入设备树前，内核中有很多board-\*或plat\*文件用于描述板卡硬件信息。引入设备树之后，硬件描述信息全部转移到设备树中，而且使用设备树还有一个好处就是，硬件信息变化时，无需再编译内核，只需要单独编译设备树。设备树也可以包含对驱动的配置信息，这样可以使驱动更灵活，在需要调整时，可以直接更改设备树。

# DTS、DTB、DTSI、DTC的关系

一个dts对应一个设备，我们最常更改的也基本上就是.dts文件，我们可以通过DTC将dts文件编译为最终使用的二进制文件dtb。

因为同一个SoC的信息是不会变化的，而我们使用同一个SoC可以做出多种使用不同器件的产品，所以每个SoC基本都会存在.dtsi文件，将SoC的公共信息进行提炼，当然我们也可以将我们自己重复使用的设备信息提炼为.dtsi文件。

与C语言一样，dts文件可以使用#include来引用头文件，可以引入.h、.dts、.dtsi。

# 设备树格式

设备树中通过节点描述设备信息，格式如下：

[label:] node-name[@unit-address] {

[properties definitions]

[child nodes]

};

[]中的部分非必须项。

label是设备别称，可以用于方便的对节点进行访问，无需使用命令比较繁琐的设备节点名称去访问，而是直接通过简略的别称进行访问。比如想访问mmc1: mmc@48060000节点，只需使用&mmc1即可。

node-name是设备节点名称，根节点用”/“表示。unit-address一般指设备地址或者寄存器地址。

{}里就是节点的内容，节点内容中包括节点属性以及此设备的子节点，子节点的格式也是如上面描述的一样。下面我们看节点的属性。

如下是am33xx的mmc1的设备树描述：

mmc1: mmc@48060000 {

compatible = "ti,omap4-hsmmc";

ti,hwmods = "mmc1";

ti,dual-volt;

ti,needs-special-reset;

ti,needs-special-hs-handling;

dmas = <&edma\_xbar 24 0 0

&edma\_xbar 25 0 0>;

dma-names = "tx", "rx";

interrupts = <64>;

interrupt-parent = <&intc>;

reg = <0x48060000 0x1000>;

status = "disabled";

};

节点的属性分为4种形式：

1. property-name;

属性为空值，如上面的ti,dual-volt。

1. property-name = <arrays of cells>;

用<>括起来的值内容是 32 位数据的合集，如上面的interrupts = <64>和reg = <0x48060000 0x1000>。

1. property-name = “string”;

用””括起来的表示字符串，如上面的compatible = "ti,omap4-hsmmc"。

1. property-name = [bytestring];

用[]括起来的表示字符序列，没用到过。

下面介绍各种节点属性的具体含义：

## compatible 属性

compatible 属性也叫兼容性，他的值为字符串列表，是设备与驱动匹配的关键。兼容性可以包含多个字符串，用于兼容如寄存器兼容的不同芯片等。兼容性又分为根节点的兼容性和设备节点的兼容性。

根节点的兼容性的字符串一般包含板子级别的名称以及芯片级别的名称，如am335x的开发板的根节点兼容性为compatible = "ti,am335x-evm", "ti,am33xx"，驱动中的DT\_MACHINE\_START的.dt\_compat就包含了设备树中的根节点兼容性，如果两者匹配，就会一系列初始化函数的执行，如下为am33xx的DT\_MACHINE\_START：

static const char \*const am33xx\_boards\_compat[] \_\_initconst = {

"ti,am33xx",

NULL,

};

DT\_MACHINE\_START(AM33XX\_DT, "Generic AM33XX (Flattened Device Tree)")

.reserve = omap\_reserve,

.map\_io = am33xx\_map\_io,

.init\_early = am33xx\_init\_early,

.init\_machine = omap\_generic\_init,

.init\_late = am33xx\_init\_late,

.init\_time = omap3\_gptimer\_timer\_init,

.dt\_compat = am33xx\_boards\_compat,

.restart = am33xx\_restart,

MACHINE\_END

设备节点兼容性用于匹配设备与驱动，匹配成功后会引发驱动probe函数的执行。驱动中会定义of匹配表用于与设备节点的兼容性匹配。以下为TI的davinci mcasp的platform\_driver的匹配表，如果设备树中定义了mcasp\_dt\_ids列表中的任何一个，就会引发davinci\_mcasp\_probe的执行：

static const struct of\_device\_id mcasp\_dt\_ids[] = {

{

.compatible = "ti,dm646x-mcasp-audio",

.data = &dm646x\_mcasp\_pdata,

},

{

.compatible = "ti,da830-mcasp-audio",

.data = &da830\_mcasp\_pdata,

},

};

static struct platform\_driver davinci\_mcasp\_driver = {

.probe = davinci\_mcasp\_probe,

.remove = davinci\_mcasp\_remove,

.driver = {

.name = "davinci-mcasp",

.of\_match\_table = mcasp\_dt\_ids,

},

};

驱动中为了兼容不同的设备，可以通过判断兼容性的内容（of\_device\_is\_compatible）来进行一些差异化的操作，当然也可以通过上面代码中的.data属性来定义差异化的数据，代码中通过of\_match\_node()或者of\_match\_device()取到兼容性对应的of\_device\_id结构，如果定义为match，则match->data即为与兼容性对应的私有数据，这样可以避免大量的判断，易于代码的扩展。

关于上面提到的设备树与驱动的匹配过程我将会在platform driver中展开来讲。

## model 属性

model 属性的值也是字符串，一般用来表示板子的名称，比如am335x的model = "TI AM335x EVM"，驱动只是用于打印信息。

pr\_info("Machine model: %s\n", of\_flat\_dt\_get\_machine\_name());

## 地址属性

设备树中#address-cells和#size-cells属性分别代表当前节点子节点的reg属性中使用多少个u32整数来表示address和length。

reg的格式为reg = <address1 length1 address2 length2 address3 length3..>，上面提到的mmc1的设备树描述中的reg = <0x48060000 0x1000>就表示mmc1的寄存器地址为0x48060000，长度为0x1000。

## device\_type 属性

此属性用于cpu节点和memory节点，在 cpu 节点中 device\_type = “cpu”，在 memory 节点中 device\_type = “memory”。device\_type也可以为"dma"、"pci"等。

## 中断节点属性

如果为中断控制器，设备树节点包含如下属性：

interrupt-controller; 表示它为一个中断控制器。

#interrupt-cells = <x>; 表示引用此中断控制器时需要多少个cell。

1. #interrupt-cells = <1>时，表示引用此节点时，只需要一个cell用于表示中断号。
2. #interrupt-cells = <2>时，表示引用此节点时不仅要指明中断号还需要指明中断的触发类型，第二个cell的bits[3:0] 用来表示中断触发类型：

1 = low-to-high edge triggered，上升沿触发 IRQF\_TRIGGER\_RISING

2 = high-to-low edge triggered，下降沿触发 IRQF\_TRIGGER\_FALLING

4 = active high level-sensitive，高电平触发 IRQF\_TRIGGER\_HIGH

8 = active low level-sensitive，低电平触发 IRQF\_TRIGGER\_LOW

在设备节点中，通过interrupt-parent指定其依附的中断控制器，如果未指定则从父节点继承。Interrupts用于指定中断号和触发类型，上面已介绍。

对于使用多个中断号的设备，如TI的mcasp控制器的设备树中中断属性如下：

interrupts = <80>, <81>;

interrupt-names = "tx", "rx";

驱动中就可以通过platform\_get\_irq\_byname(pdev, "rx")来获取对用的IRQ。

## GPIO节点属性

GPIO控制器的设备树节点定义如下：

gpio0: gpio@44e07000 {

compatible = "ti,omap4-gpio";

ti,hwmods = "gpio1";

gpio-controller;

#gpio-cells = <2>;

interrupt-controller;

#interrupt-cells = <2>;

reg = <0x44e07000 0x1000>;

interrupts = <96>;

};

#gpio-cells = <2> 引用此控制器时第一个cell为GPIO号，第二个cell为GPIO的级性。为1(GPIO\_ACTIVE\_LOW)时表示低电平有效，为0(GPIO\_ACTIVE\_HIGH)时表示低电平有效。

引用GPIO时可以使用cd-gpios = <&gpio0 18 GPIO\_ACTIVE\_HIGH>进行引用，内核提供了很多获取gpio的API：

获取gpio:

int of\_get\_named\_gpio(struct device\_node \*np, const char \*propname, int index)

其中propname即为上面的“cd-gpios”

int of\_get\_gpio(struct device\_node \*np, int index)

获取gpio数目

int of\_gpio\_named\_count(struct device\_node \*np, const char\* propname)

int of\_gpio\_count(struct device\_node \*np)

对于上面的of\_get\_gpio()和of\_gpio\_count()无需指定propname，默认使用"gpios"，所以使用这两个函数时设备树中的gpios名称必须为gpios = <&gpio0 18 0>。

## 时钟节点属性

## pinmux节点属性

# 常用of函数

内核中定义了大量的of函数，上面已略有提及，of函数的定义在include/linux/of.h 中，实现则是在drivers/of中，下面介绍一些常用的，可以慢慢的丰富。

## 节点查找

struct device\_node \*of\_find\_node\_by\_name(struct device\_node \*from, const char \*name);

通过节点名称查找节点。

from：从这个节点开始查找，输入 NULL 时从根节点开始查找。

name：目标节点名称。

返回值：找到目标节点迒回 device\_node 绌极体。没有找到时迒回 NULL

struct device\_node \*of\_find\_node\_by\_type(struct device\_node \*from, const char \*type);

通过device\_type 属性查找节点。

struct device\_node \*of\_find\_compatible\_node(struct device\_node \*from, const char \*type,const char \*compatible);

通过 device\_type 和compatible 属性查找节点。

type：device\_type 属性值， 输入 NULL 时忽略。

struct device\_node \*of\_find\_node\_by\_path(const char \*path);

通过节点路径查找节点。

path:节点的完成路径，比如查找memory节点则使用of\_find\_node\_by\_path("/memory")。

## 属性读取

int of\_property\_read\_u32\_array(const struct device\_node \*np,const char \*propname, u32\*out\_values, size\_t size);

获取含有多个属性值的多个数据，其中的u32表示数据类型为u32，也可以更改为u8,u16,u64。

propname：目标属性名。

out\_values：读取到的数据指针。

size：要读取的数据数量。

返回值：

0：读取成功；

-EINVAL：属性不存在；

-ENODATA：属性无数据；

-EOVERFLOW：属性值数据数量小于 size。

int of\_property\_read\_u32(const struct device\_node \*np,const char \*propname, u32 out\_value);

获取只有单个值的属性数据，同样u32可以替换为u8,u16,u64。

函数参数与返回值同上。

int of\_property\_read\_string(struct device\_node \*np, const char \*propname, const char \*\*out\_string);

获取属性中字符串值。

bool of\_property\_read\_bool(const struct device\_node \*np, const char \*propname)

查找节点是否存在对应属性。

## 内存映射

## 中断解析

## platfrom设备

# 测试程序说明

设备树中添加如下节点：

dts\_test: dts\_test@0 {

compatible = "dts-test";

dts-test-bool;

dts-test-u32 = <0x12345678>;

dts-test-string = "hello dts";

dts-test-u32-array = <0x12 0x34 0x56 0x78>;

};

测试结果如下：

